CMPE160 – Introduction to Object Oriented Programming

Project #4 – Blockchain Implementation

Deadline: 30.06.2020 – 23:59

1. Introduction

Blockchain, as the name implies, is a chain of blocks is used as a storage mechanism of records. Even though it becomes popular after the usage for cryptocurrency, it has the potential of being used for the business. Supply chain management, financial operations and many other areas are suitable for blockchain to be integrated into.

The most basic implementation of a blockchain consists of two different data structures: (1) Linked list and (2) Merkle tree. You all know what a linked list is. In blockchain, linked list forms the chain of the blocks. However, this linked list is a little bit different than the LinkedList class provided by JCF. In the list of blocks, each block points to the previous block. While it is possible to traverse over a LinkedList object in both directions, blockchain does not allow such operations and it is not possible to insert a block in an arbitrary location.

Merkle tree is mostly used for consistency of the data. In blockchain, the data that is dealt with is the transactions. Therefore, in order to provide the consistency of the transactions and prevent any fault or fake transactions to be inserted into the blockchain, Merkle tree is used.

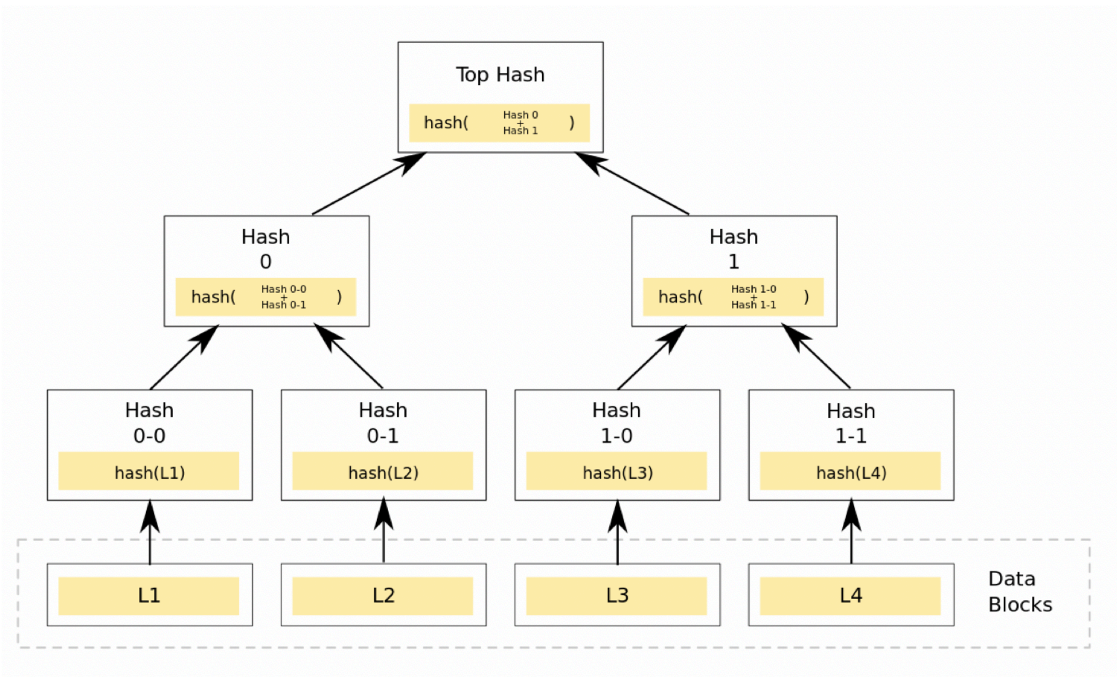

Figure1: An illustration of a Merkle Tree

Here is an example of a Merkle tree, as depicted in Figure 1. The leaf nodes of a Merkle tree represents the hash of a particular data block. On the other hand, the non-leaf nodes store the hash of the data stored in the child nodes. As an example, the leftmost leaf node stores the hash value of the data block L1 (Hash0-0 = hash(L1)), and its sibling stores the hash value of the data block L2 (Hash0-1 = hash(L2)). The parent of these leaf nodes stores the hash of the concatenation of the data stored in the child node, which is the hash of L1 and hash of L2 (`Hash 0 = hash(concatenate(Hash0-0, Hash0-1))`).

The advantage provided by the usage of Merkle tree, especially in blockchain, is to detect any inconsistency among the data. If there is any, by traversing over the tree, one can find the source of the problem by comparing the calculated hash in the tree and the actual hash.

In this project, you are going to implement a basic blockchain mechanism with the data structures mentioned above. The following section provides the detailed explanation of how linked list and Merkle tree cooperate to function as a blockchain. Additionally, the implementation details with the requirements are also discussed.

2. Implementation Details

As specified, blockchain mainly consists of a linked list in which each block points to the previous block. Any time a block is added into the list, a reference to the previous block is create immediately.

Every single block in the chain (or list) stores the ultimate hash of the previous block's Merkle tree. In other words, each block is associated with a Merkle tree, and the value stored in the root of a Merkle tree is also stored in the next block.

Even though there are multiple versions of Merkle tree, in this project you are going to use binary version of it. Besides, the capacity (or the number of transactions hashed in a Merkle tree) of a Merkle tree may vary among applications. In this project, each Merkle tree of a block contains exactly 4 transactions. For example, Transactions from 1 to 4 is stored in the first block, transactions from 5 to 8 is stored in the second block and so on. As soon as the (totalNumberOfTransactions % 4) becomes 1, it means that a Merkle tree can be created for a new block in the chain. However, please keep in mind that the most recently added block can be associated with less than 4 transactions.

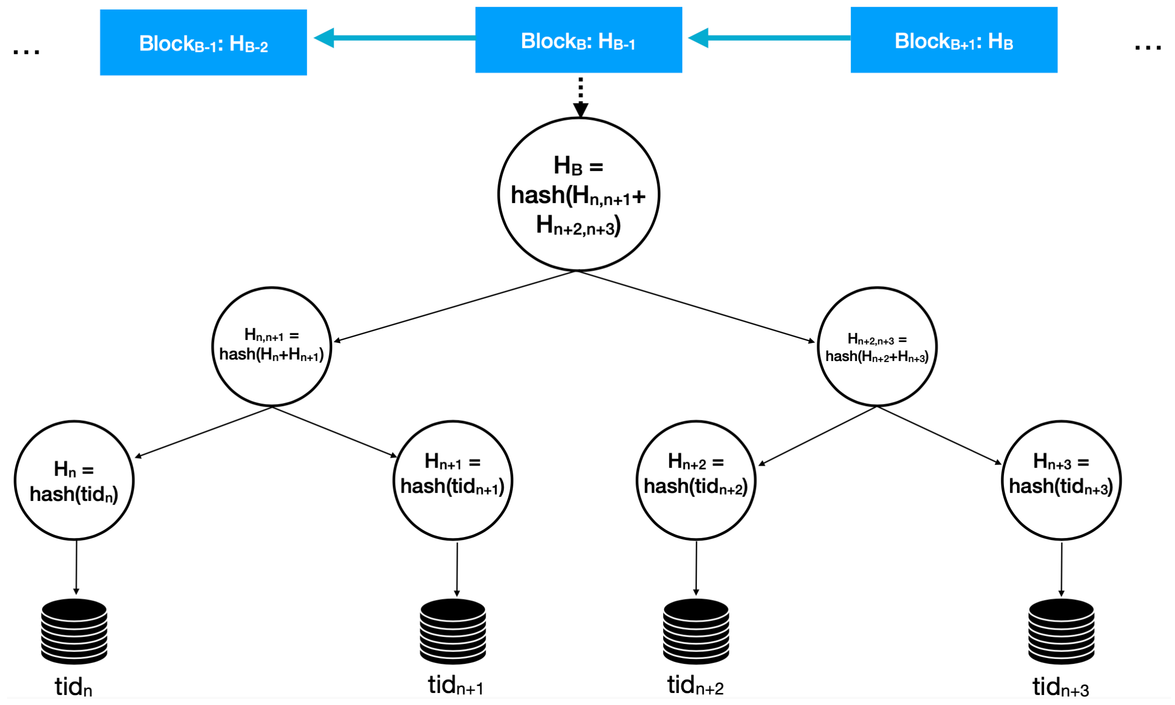
...
$Block_{B-1}: H_{B-2}$
$Block_B: H_{B-1}$
$Block_{B+1}: H_B$
...
$H_B = hash(H_{n,n+1} + H_{n+2,n+3})$
$H_{n,n+1} = hash(H_n+H_{n+1})$
$H_{n+2,n+3} = hash(H_{n+2}+H_{n+3})$
$H_n = hash(tid_n)$
$H_{n+1} = hash(tid_{n+1})$
$H_{n+2} = hash(tid_{n+2})$
$H_{n+3} = hash(tid_{n+3})$
tid_n
tid_{n+1}
tid_{n+2}
tid_{n+3}

Figure 2. Blockchain Structure in the Project

In Figure 2, you can see the illustration of the blockchain that you are going to implement in this project. In $Block_B$, the hash value available in the root of the previous block is stored. Same applies for the other blocks in the chain. In the linked list implementation, as stated, each block points to the previous block (reverse of the traditional implementation).

In the Merkle tree of $Block_B$, there are 4 different transactions, with their IDs. The ID of the transaction(n) is notated as tid_n. The similar hashing operations shown in Figure 1 also apply for this implementation.

SHA2 hashing algorithm family is the mostly used one for the hashing operations in a Merkle tree. Accordingly, you are going to use SHA-256 hashing algorithm (256 bits) in this project. The methods of hashing are already provided for you in the `HashGenerator` class, there is no need for extra implementation to calculate hashes in the Merkle tree. In this project, the hash values will be in the form of String. While the legacy approach of SHA-256 in Java generates hash result as an array of bytes, the `String byteArrayToString(byte[] hashInByte)` method converts it to the String, which is then returned by the `String generateHash(String data)` method.

Within the scope of this project, you must implement the following requirements, with the exact class, field and method names:

- `Transaction class`
  - The transaction class represents the necessary information about a transaction. In this project, we are not going into the details of a transaction. The only necessary information to be stored is `String transactionID`.
  - This class should be implemented with a customized constructor: `Transaction(String transactionID)`, which initializes the `transactionID` field with the value passed through the argument.
- `Block class`
  - This class represents the components of the blockchain. Each block instance should have the following fields:
    - `String hashValue`, which stores the hash of the root of the previous block
    - `Node root`, which is a reference to the root node of its own Merkle tree
    - `Block prev`, which is a reference to the previous node in the blockchain
- `Node class`
  - This class represents the nodes in a Merkle tree. The fields of the `Node` class are as follows:
    - `String data`, which stores the corresponding hash value
    - `Node left`, `Node right`, which are the references to the child nodes
- `LeafNode class` (subclass of the `Node`)
  - This class should `extend` the `Node` class. The leaf nodes are represented through this class. The following fields are required:
    - `Transaction t`, which is a reference to the transaction to be hashed
    - `Node left` and `Node right` references should be `null`
- `Blockchain class`
  - This is the main blockchain class. The list of fields of this class is:
    - `Block recentBlock`, which is a reference to the most recently added block to the chain
  - The constructor of the Blockchain class should be defined as `Blockchain()`, which initializes the `recentBlock` field as `null`.
  - Here is the list of methods that must be implemented in this class:
  - `void addBatchTransactions(String listOfTransactions)`
    - The string value passed as the parameter is actually name of a file that should be read by your program. This file is composed of the IDs of a set of transactions that are executed. In this file, each line represents a particular transaction. The ID of a transaction is composed of 16 alphanumeric characters. After the constructor is executed, your program should construct the blocks, trees, and entire blockchain. Lastly, the `recentBlock` field should be assigned correctly to point the corresponding block in the chain.
  - `void addSingleTransaction(String transactionID)`
    - The string value passed as the parameter is a new `transactionID`. This method should find the corresponding place for this transaction and manipulate the Merkle trees and blocks to embed this transaction into the blockchain.
  - `ArrayList<Stack<String>> validate(String correctHashList)`

- The string value passed as the parameter is the name of another file that should be read by your program. This file includes a set of hash values, organized as line by line. The organization of this text file can be explained as follows:
  - If there are 15 transactions in total (t1…t15), it means that there are 4 blocks (B1, B2, B3, B4, where B4 is the most recently added one) in total. In other words, there should be 4 different Merkle tree instances.
  - The `correctHashList` stores the correct hash values of each node in the BFS order starting from B1. Let us explain it in this way: the first 7 lines of the text file includes the hash values of the nodes in the Merkle tree of B1 in BFS order (since there are 7 nodes in the tree). Then, the second 7 lines includes the hash values of the nodes in the Merkle tree of B2 in BFS order. Then, the third 7 lines includes the hash values of the nodes in the Merkle tree of B3 in BFS order. Lastly, last 6 lines (since there are 15 transactions, there are only 3 transactions in the B4) contains the hash values of the nodes in the Merkle tree of B4 in BFS order.
- This method compares the blockchain that you build, and the correct hash values that should be in a correctly built blockchain. A `transactionID` passed to the program may be fake, or manipulated. Therefore, we must ensure that the blockchain is validated.
- This method returns an instance of `ArrayList<Stack<String>>.` In fact, each slot of the `ArrayList` is for an invalid `transactionID` integrated to the blockchain. Each index of the `ArrayList` is an object of `Stack<String>.` For example, let us assume that the ID of the Transaction 1 is fake. This results in that the root of the Merkle tree of Block 1 and the value stored in Block 2 are wrong. Now we can observe the advantage of Merkle tree usage in blockchain. Instead of traversing every single node in the trees, we can exploit the hashing comparison. For example, if tid_{n+2} is fake in Figure 2, then H_B is not correct. After finding that there is a problem in this particular tree (though the value stored in $Block_{B+1}$), we first compare the $H_{n, n+1}$ with the correct one. If it is true, then we don't need to check its child nodes. On the other hand, $H_{n, n+1}$ becomes validated means that $H_{n+2, n+3}$ is invalid because the root is invalid. Therefore, only the right subtree is traversed for finding the invalid transaction. This enhances the performance of the search algorithm. Thus, a `Stack<String>` instance in an index of `ArrayList` should store the list of hash values traversed to find the invalid transaction. In other words, it shows us the path used for finding the invalid transaction. In our example, `Stack<String>` should include in this exact order: (the first element is the first one pushed into the stack) H_B - $H_{n+2, n+3}$ - H_{n+2}. Since we are starting from the root, first H_B is pushed into the stack.
- The validation starts from the most recent block added to the chain and it traverses the blockchain block by block, until the oldest block is inspected. Therefore, the indexes of the `ArrayList` stores the invalid transactions in the reverse order they are added to the tree (except the blocks with multiple invalid transactions. In that case, order within a block is not important).
- If there is not any invalid transaction within the blockchain, this method should return an empty `ArrayList`.

3. Important Remarks

- You can define your own fields, methods and classes as you desire.
- The decision upon the access modifiers are up to you. However, you need to select the appropriate modifiers according to the case and avoid using public for everything.
- You need to document your code. %5 of your project grade will be dedicated to the documentation.
- Late submissions are not allowed.